Evolving behaviour trees for swarm robotics

Simon Jones, Matthew Studley, Sabine Hauert, Alan Winfield

Abstract Controllers for swarms of robots are hard to design as swarm behaviour emerges from their interaction, and so controllers are often evolved. However, these evolved controllers are often difficult to understand, limiting our ability to predict swarm behaviour. We suggest behaviour trees are a good control architecture for swarm robotics, as they are comprehensible and promote modular reuse. We design a foraging task for kilobots and evolve a behaviour tree capable of performing that task, both in simulation and reality, and show the controller is compact and understandable.

1 Introduction

Swarm robotics is the field of robotics inspired by social insects, flocks of birds, schools of fish and other natural collective phenomena. By using many simple and cheap robots, it is hoped that goals such as pollution control, mapping and exploration, and disaster recovery could be met in ways which are resilient, scalable and decentralised [1]. The desired collective behaviour of the swarm emerges in a self-organised way from the interactions of the many individual agents that make up the swarm. Designing the controller for these agents is notoriously hard. A commonly used approach is the use of evolutionary methods to discover suitable controller designs.

Behaviour trees are widely used in the games industry to represent the decision processes of non-player characters. Recently, they have been applied to robotics, although not to our knowledge to swarm robotics. They have desirable properties that make them interesting to consider in the context of swarm robotics. They are human readable. They are hierarchical, all subtrees are themselves behaviour trees,

Simon Jones simon.jones@brl.ac.uk
Matthew Studley matthew2.studley@uwe.ac.uk
Sabine Hauert sabine.hauert@bristol.ac.uk
Alan Winfield alan.winfield@uwe.ac.uk
Bristol Robotics Laboratory, University of the West of England, Bristol, UK

encapsulating a complete behaviour that can exist within a larger tree, offering possibilities for modularity and building block reuse. Finally, they can be created and optimised using the techniques of Genetic Programming [2].

In this work, we design a behaviour tree controller architecture suitable for instantiation in a swarm of kilobots. We then automatically evolve behaviour trees in simulation to enable the swarm to perform a collective foraging task. The fittest behaviour tree is then evaluated in a swarm of real robots and analysed.

This paper is organised as follows; Section 2 gives a brief overview of swarm robotics and the kilobot platform, and introduces behaviour trees, Section 3 describes the experimental procedure, Section 4 details the results and Section 5 discusses results and possible further work.

2 Background and Previous work

We work within the paradigm of swarm robotics as described by Şahin [3] taking inspiration from social insects, where many simple, homogeneous and not particularly capable robots with only local sensing and knowledge interact to produce a desired collective behaviour. There are no principled solutions to designing the controller to produce a given collective behaviour, common approaches are based on bioinspiration, evolutionary methods and gaining insight by reverse engineering the discovered controllers [4–6]. See [7] for a recent survey of the state of automatic swarm controller generation.

One problem with automatic generation of swarm controllers is that of bootstrapping; it is difficult to devise fitness functions to get complex behaviours [8, 9], the evolutionary process will often get stuck in uninteresting local maxima. Iterative approaches, with a gradually complexifying fitness function can work well, but this requires the designer to *a priori* specify the path to the eventual complex behaviour, lessening the likelihood of discovering novel behaviours. Hierarchical modular approaches are a promising alternative. AutoMoDe by Francesca *et al.* [10, 11] uses hand-designed modular and parameterised sub-behaviours which are combined within a Probabilistic Finite State Machine (PFSM), and the module parameters and PFSM topology constitute a search space over which optimisation is automatically carried out. Interestingly their automatically generated controllers have a lower reality gap compared to pure neural net approaches. Another modular approach is work by Duarte *et al.* [12,13] where individual sub-behaviours are separately evolved neural net controllers which are again combined in a higher level Finite State Machine (FSM), this time hand-designed.

A behaviour tree (BT) is a hierarchical structure of nodes, with leaves that interact with the state of the world, and inner nodes that link these actions together in various conditional and sequential ways. The whole tree is evaluated at regular intervals, this is termed a *tick*. The *tick* is propagated down to the leaves and results are propagated back up according to the node types. Ogren [14] shows that all Hierarchical Dynamic Systems and therefore Finite State Machines (FSMs) can be represented by a BT, provided there are both sequence and selection type operators. With the addition of a probabilistic selector, Probabilistic Finite state Ma-

chines (PFSMs) can also be represented. Compared to an FSM or PFSM, the state transitions are implicit in the tree structure, and modular[1] structure is explicit; all subtrees are legal behaviour trees. Behaviour trees have their origins as a graphical software engineering tool before being adopted by the games industry for describing the decision processes and actions of non-player characters. Recently they have been formalised and applied to robotics [14, 16–30].

Kilobots are small cheap robots introduced by Rubenstein *et al.* [31]. They are capable of motion using two vibrating motors, communication with each other over a limited range using IR, distance sensing using the communication signal strength, environmental sensing with an upwards facing photo detector, and signalling with a multicolour LED. They are cheap enough to make it practical to build very large swarms and capable enough to run interesting experiments. Collective control of the kilobots in order to program and to start or stop them is achieved using a high intensity IR system using the same protocol as the inter-kilobot communication system.

3 Materials and Methods

Foraging as a collective task is often used as a benchmark for swarm systems [32]. It involves robotic agents leaving a nest region, searching for food, and returning food to the nest. Cooperative strategies are often more effective.

We designed a simple foraging experiment for a swarm of kilobots in an arena upon which we can project patterns of light to define the environment (Fig. 1). At the centre of the arena is a circular *nest* region. Surrounding this is a gap, then beyond that is the *food* region. A kilobot which moves into the food region is regarded as having picked up an item of food, a kilobot which is carrying an item of food that enters the nest region is regarded as depositing the food in the nest. Multiple kilobots are placed in the central region in a grid and all execute the same controller (homogenous swarm) for a fixed amount of time. The fitness of the swarm is related to the total amount of food returned to the nest within the test time. The maximum

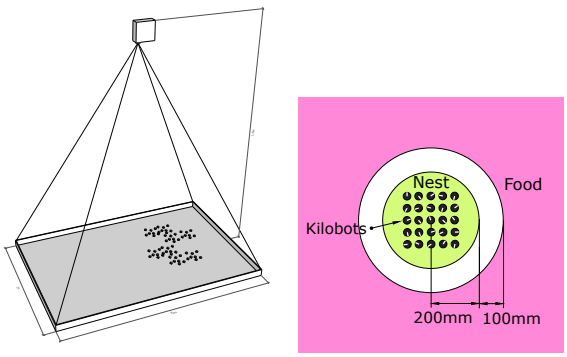

Fig. 1 Left: Kilobot arena. The arena is a 3m x 2m surface upon which a projector defines the environment with patterns of light. Right: Starting configuration for kilobot foraging experiment. 25 kilobots are placed in a 5x5 grid in the centre of the nest region, with random orientations. Surrounding the nest is a 100mm gap, then outside that is the food region.

possible number of food items depends on the starting spatial distribution of the

[1] Perhaps mirroring a fundamental property of nature [15].

kilobots. Assume that the kilobots start on the edge of the *nest* region and for the duration of the test move directly back and forth between *nest* and *food* regions by the shortest distance. Let $food_{max}$ be the maximum food items, t_{test} be the test time, v_{avg} be the average linear velocity of the kilobots, n be the number of kilobots, fn_{dist} be the shortest (radial) distance between the food and nest regions:

$$food_{max} = \frac{n \cdot v_{avg} \cdot t_{test}}{2 \cdot fn_{dist}} \tag{1}$$

We normalise the actual collected food items within the time of the test to give a fitness value. Let $food_{collected}$ be the total collected food items and k be a derating factor. The fitness f of the controller is given by:

$$f = k \cdot \frac{food_{collected}}{food_{max}} \tag{2}$$

The derating factor k is used to exert selection pressure towards smaller behaviour trees to ensure they will fit within the limited RAM resources of the kilobots. It is related to r_{usage} (4) in the following way: $k = 1.0$ when $r_{usage} < 0.75$ decreasing linearly to 0 when $r_{usage} = 1.0$.

Kilobots. For our experiments, we want to be able to sense whether we are within a particular region (nest or food) of the arena. Regions are delineated within the arena by using different coloured light from a video projector and detected with the upwards-facing phototransistor of the kilobots. In order to create a robust region sensing capability with a monochrome sensor, we exploited some particular characteristics of low cost DLP projectors [33].

The optical path of these type of projectors consists of a white light source, an optical modulator array, and a spinning colour wheel with multiple segments. Different full intensity primary and secondary colours produce different, quite distinct brightness modulation patterns in the light, which our eyes integrate but which we can detect easily with a series of samples from the photodetector. In our case, the projector had a wheel spinning at 120 Hz. Within each 8.3 ms period, primary colours were represented with a single pulse of about 1.2 ms, cyan and yellow with a pulse of 3.5 ms, and magenta with two pulses of 1.2 ms separated by a gap of 2 ms, giving, including black, four distinguishable patterns. We take 16 brightness samples from the phototransistor at 520 us intervals, covering one complete cycle, and classify the pattern.

The IR communication system between the kilobots has a range of about 100 mm. Twice a second, the kilobot system software sends any available outgoing message, retrying if the sending attempt collided with another sender. A kilobot receiving a valid message calls a user specified function to handle it. The message has a payload of nine bytes, and associated with the message is signal strength information to enable the distance from the sender to be calculated.

Controller. In order to control a robot with a behaviour tree, we need to define the interface between the behaviour tree action nodes and the robot, and the action

nodes that act on the interface. This interface is known as the *blackboard*. Here there is a trade-off between the capabilities that we choose to hard code and those that we hope will evolve in the BT. We do not design the behaviour of the swarm but we do make assumptions about what kind of sensory capabilities might be useful for the evolutionary algorithm. This is often implicit in swarm robotics. The kilobot has no in-built directional sensors, like the range-and-bearing sensors that are common in swarm robotics experiments, so we synthesise collective sensing such that it is possible for a robot to tell if it is moving towards or away from the food or nest. We also give the capability of sensing the environment and the local density of kilobots, and of sending and receiving signals to other kilobots.

This relatively rich set of hardwired capabilities is outlined in Table 1. There are ten blackboard entries, **motors** maps to the motion control commands of the kilolib API, The **send_signal** and **receive_signal** entries allow for communication between

Index	Name	Access	Description
0	$motors$	W	0=off, 1=left turn, 2=right turn, 3=forward
1	$scratchpad$	RW	Arbitrary state storage
2	$send\_signal$	RW	>0.5 = Send a signal flag
3	$received\_signal$	R	1=A signal flag has been received
4	$detected\_food$	R	1=Light sensor showing food region
5	$carrying\_food$	R	1=Carrying food
6	$density$	R	Density of kilobots in local region
7	$\Delta density$	R	Change in density
8	$\Delta dist_{food}$	R	Change in distance to food
9	$\Delta dist_{nest}$	R	Change in distance to nest

Table 1: Behaviour tree blackboard, defining interface between the behaviour tree and the robot.

kilobots initiated within the BT; **send_signal** is writeable from the BT. When the value is greater than 0.5, it is considered true, and a signal flag will be set in the stream of outgoing message packets. The **receive_signal** entry will be set to 1 if any message packets were received over the previous update cycle that had their signal flag set, otherwise it will remain zero. The **scratchpad** can be read and written, and has no defined meaning, it makes available some form of memory for the evolution of the BT to exploit. **Detected_food** is read-only, and is 1 if the environment sensing shows that the kilobot is in the food region, and zero otherwise, and **carrying_food** denotes whether the kilobot is considered to be carrying a food item. This entry is set to 1 if the kilobot enters the food region, and cleared to zero if the kilobot enters the nest region.

The remaining four entries are all metrics derived from the incoming stream of messages and their associated distance measurements. $density$ and $\Delta density$ are measures of the local population density and how it is changing. Each kilobot has a unique ID, which is embedded in its outgoing message packets. By tracking the number of unique IDs and the distances associated with messages from them, we can estimate the local density. Let $UID_{received}$ be the set of unique IDs received in the last update cycle, $dist_i$ be the distance in mm associated with the unique ID, the raw local density in $kilobots \cdot m^{-2}$ in an update cycle d_{raw} is given by:

$$d_{raw} = \sum_{i \in UID_{received}} \frac{1}{\pi(dist_i/1000)^2} \quad (3)$$

This value is filtered with a moving average over $w = 5$ update cycles[2] to give $density(t)$ at update cycle t and $\Delta density(t) = density(t) - density(t-1)$.

The two distance metrics $\Delta dist_{food}$ and $\Delta dist_{nest}$ are calculated by tracking the minimum communication hops [34] needed to reach the respective region, illustrated in Fig. 2. For both food and nest, within the message packet are two fields, a hop count and an accumulated distance. The hop count is the minimum number of message hops to reach either the food or the nest region. The accumulated distance is the total length of those hops. Kilobots receiving messages select the lowest hop count, increment it and forward it and the new accumulated distance in the outgoing message stream. If no messages are received, we default to a distance of 0 mm if in a food or nest region, or 500 mm if not in a region. At every update cycle, we calcu-

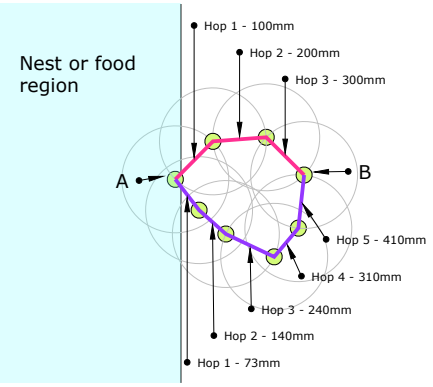

Fig. 2 Calculation of distance metrics. Kilobot 'A' is in a food or nest region, kilobot 'B' is connected to 'A' via two routes. Grey circles denote maximum communications radius. 'B' selects the message from the top route because the hop count is lowest, giving an accumulated distance along hops to the region of 300mm.

late two raw distance measures $dist_{food\_raw}$ and $dist_{nest\_raw}$. These are then filtered with a moving average in the same way as the *density* value.

The behaviour tree nodes we implement are outlined in Table 2. Nodes are divided into two types; composition and action. Composition nodes are always inner nodes of the tree and combine or modify the results of subtrees in various ways. Action nodes are always leaf nodes and interface with the blackboard. Every update cycle, occurring at 2 Hz, the root node of the tree is sent the *tick* event. Each node handles the *tick* according to its function and returns *success*, *failure*, or *running*. The propagation of *tick* events down the tree and the return of the result to the root happen every cycle. The composition nodes **seqm, selm, probm** can have either 2, 3, or 4 children. On receiving a *tick* they process their child nodes in the following way: **seqm** will send *tick* to each child in turn until one returns *failure* or all children have been *tick*ed, returning *failure* or *success* respectively, **selm** will send *tick* to each child in turn until one returns *success* or all children have been *tick*ed, returning *success* or *failure* respectively, **probm** will probabilistically select one child node to send *tick* to and return what the child returns. They all have memory, that

[2] Chosen in simulation as a reasonable compromise between responsiveness and stability

is, if a child node returns *running* the parent node will also return *running*, and the next *tick* event will start from that child node rather than the beginning of the list of child nodes. The **repeat, successd, failured** nodes have a single child. **repeat** sends up to a constant number of *tick*s to its child for as long as the child returns *success*, **successd** and **failured** send *tick* to their child and then always return *success* or *failure* respectively. The action nodes are leaf nodes and interface with the blackboard, described in Table 1. **ml, mr, mf** turn left, right, or move forward, returning *running* for one cycle, then *success*. The various **if** nodes compare blackboard entries with each other or with a constant, and the **set** node writes a constant to a blackboard entry.

Node	Size	*success* if	*failure* if	*running* if	Description
Composition nodes					
seqm2,3,4	7,9,11	$N\ Ch\ S$	$1\ Ch\ F$	$1\ Ch\ R$	Sequence, *tick* until *failure*
selm2,3,4	7,9,11	$1\ Ch\ S$	$N\ Ch\ F$	$1\ Ch\ R$	Selection, *tick* until *success*
probm2,3,4	11,17,23	$Ch_r\ S$	$Ch_r\ F$	$Ch_r\ R$	Probabilistic choice
repeat	6	$I\ Ch\ S$	$1\ Ch\ F$	$Ch\ R$	Repeat subtree I times
successd	4	$Ch\ \bar{R}$	*never*	$Ch\ R$	Always succeed subtree
failured	4	*never*	$Ch\ \bar{R}$	$Ch\ R$	Always fail subtree
Action nodes					
mf	2	$t = 1$	*never*	$t = 0$	Move forward for 1 *tick*
ml	2	$t = 1$	*never*	$t = 0$	Turn left for 1 *tick*
mr	2	$t = 1$	*never*	$t = 0$	Turn right for 1 *tick*
ifltvar	4	$v_1 < v_2$	$v_1 \geq v_2$	*never*	If $v_1 < v_2$
ifgevar	4	$v_1 \geq v_2$	$v_1 < v_2$	*never*	If $v_1 \geq v_2$
ifltcon	7	$v < k$	$v \geq k$	*never*	If $v < k$
ifgecon	7	$v \geq k$	$v < k$	*never*	If $v \geq k$
set	7	*always*	*never*	*never*	Set $w \leftarrow k$
successl	2	*always*	*never*	*never*	Always succeed
failurel	2	*never*	*always*	*never*	Always fail

Table 2: Behaviour tree nodes. $Ch \equiv children$, $S \equiv succeeded$, $F \equiv failed$, $R \equiv running$, $N \equiv num\ children$, $I \equiv repeat\ iterations$, $r \equiv randomly\ selected\ child$, $t \equiv ticks$, $v,w \equiv blackboard\ entry$, $k \equiv contant$. Notation from [28].

The controller runs an update cycle at 2Hz. Message handling takes place asynchronously, and a message is always sent at each sending opportunity. Environmental sensing takes place at 8Hz, synchronously with the update cycle, with a median filter over 7 samples to remove noise. Each cycle, the following steps take place: 1) New blackboard values are calculated based on the messages received and the environment. 2) The behaviour tree is *tick*ed, possibly reading and writing the blackboard. 3) The movement motors are activated, and the message signal flag set according to the blackboard values.

Implementation of the behaviour tree for execution on the kilobot required careful use of resources; the processor has only 2kbytes RAM, which must hold all variables, the heap, and the stack. The tree structure is directly represented in memory, with each node being a structure with type, state, and additional type-dependent data such as pointers to children. Execution of the behaviour tree involves a recursive de-

scent following node child pointers and as such, each deeper level uses entries on the stack.

The compiled kilobot code uses about 500 bytes for all non-heap variables. We allocate 1024 bytes to the tree storage, leaving another 500 bytes for the stack and some margin for Interrupt Service Routine stack usage. Each level of tree depth uses 16 bytes of stack. Let tr_{size} be tree storage bytes and tr_{stack} be tree stack usage. The resource usage is given by:

$$r_{usage} = max(\frac{tr_{size}}{1024}, \frac{tr_{stack}}{500}) \tag{4}$$

This gives a maximum tree depth of about 30 and a maximum number of about 140 nodes at the average node size.

Evolutionary algorithm and simulator. Behaviour trees are amenable to evolution using genetic programming techniques. Using the DEAP library [35] a primitive set of strongly typed nodes were defined to represent behaviour tree nodes and their associated allowable constants. There are several types of constants: **if** and **set** $k \in [-1.0, 1.0]$, **repeat** iterations $I \in [1..9]$, **if** blackboard index $v_i \in [1..9]$, **set** blackboard index $w \in [1..2]$, **prob** probability $p \in [0.0, 1.0]$

Evolution proceeds as follows: The population of n_{pop} is evaluated for fitness by running 10 simulations for each individual, each simulation with a different starting configuration. The starting position is always a 5x5 grid with 50mm spacing in the centre of the nest region, but the orientation is randomly chosen from interval $(-\pi, \pi)$ radians. The simulation runs for 300 simulated seconds and fitness is as Eqn 2.

An elite of n_{elite} is transferred unchanged to the next generation. The remainder are chosen by tournament selection with size t_{size}. A tree crossover operator is applied with probability p_{xover} to all pairs of non-elite, then three different mutation operators are applied to the non-elite individuals. Firstly, with probability p_{mutu}, a node in the tree is selected at random and the subtree at that point is replaced with a randomly generated one. Next, with probability p_{muts}, a branch is chosen randomly and replaced with one of its terminals. Next, with probability p_{mutn} a node is picked at random and replaced with another node with the same argument types. Lastly, with probability p_{mute}, a constant is picked randomly and its value changed. Parameters are shown in Table 3.

We wrote a simple 2D simulator based on the games physics engine Box2D [36]. The physics engine is capable of simulating interactions between simple convex geometric shapes. We model the kilobots as disks sliding on a flat surface with motion modelled using two-wheel kinematics, with forward velocity of $8 \times 10^{-3} ms^{-1}$ and turn velocity of $0.55 rad\, s^{-1}$, based on measurements of 25 kilobots. Physical collisions between kilobots, and movement into and out of communication range were handled by Box2D, with an update loop running 10Hz. Simulator deficiencies were masked using the addition of noise [37]. Gaussian noise was added to linear ($\sigma = 1 \times 10^{-3} ms^{-1}$) and angular ($\sigma = 0.2 rad\, s^{-1}$) components of motion at every simulator timestep, and each kilobot had a unique fixed linear ($\sigma = 1.3 \times 10^{-3} ms^{-1}$)

and angular ($\sigma = 0.06 rad\,s^{-1}$) velocity bias added, to reproduce measured noise performance and variability of real kilobots. Message reception probability was fixed at 0.95. Simulation performance r_{acc}, measured using the methodology described in [38] on an iMac 3.2GHz machine was approximately 8×10^4.

Parameter	Value	Description
n_{gen}	200	Generations
t_{test}	300	Test length in seconds
n_{pop}	25	Population
n_{elite}	3	Elite
t_{size}	3	Tournament size
p_{xover}	0.8	Crossover probability
p_{mutu}	0.05	Probability of subtree replacement
p_{muts}	0.1	Probability of subtree shrink
p_{mutn}	0.5	Probability of node replacement
p_{mute}	0.5	Probability of ephemeral constant replacement

Table 3: Parameters for a single evolutionary run

Twenty five independent evolutionary runs were conducted, each one using the parameters in Table 3. Each individual fitness evaluation was the mean over ten simulations with different starting configurations. A total of 1.1 million simulations were run[3].

The fittest individual across the 25 separate populations was evaluated again for fitness, this time over 200 simulations with different starting configurations. This individual controller was then instantiated uniformly across a swarm of real kilobots, giving a homogenous swarm. The real kilobots were run 20 times with different starting configurations and their fitness measured.

4 Results and discussion

The results (Fig. 3) show that we have successfully evolved a behaviour tree for use as a swarm robot controller to perform a foraging task. When instantiated in a swarm of real robots, it performs similarly to the simulation, validating the applicability of using this simulator for evolving kilobot swarm controllers. The performance is slightly lower in real life (0.058) compared to the simulated (0.075) performance, this is expected due to *reality gap* [37] effects. It is worth noting this is still a good outcome, the robots are able to effectively forage.

Fitness rises fast to about 0.03 after the first generation. This is due to the fact that an extremely simple controller that does nothing except move forward will still collect some food; because of the variability of the kilobots, some will move in large arcs that leave the nest, enter the food region and return to the nest. This type of controller is easily discovered by the evolutionary algorithm, confirmed by examining the fittest controller after one generation in the fittest lineage. The kilobot paths in

[3] Due to the elitism policy, three individuals per generation are unchanged and need no fitness evaluation

simulation are shown in Fig. 4. It is noteworthy that the fittest of the 25 lineages is much fitter than the median, and the innovation seems to have been discovered around generation 30. This suggests that the evolutionary algorithm is not exploring the fitness landscape very effectively, otherwise we would expect evolution to discover similar behavioural innovations within other lineages.

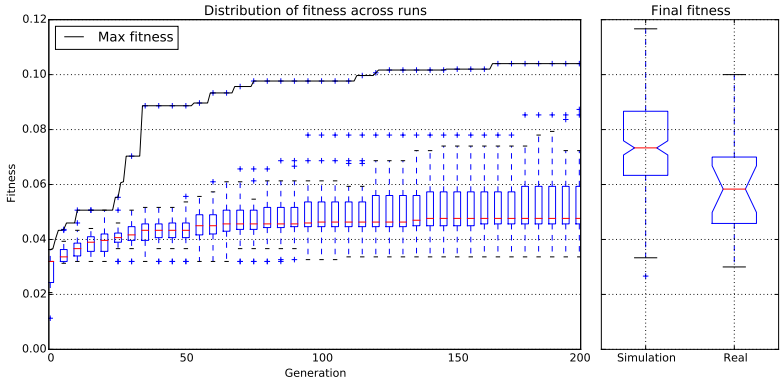

Fig. 3: Result of evolutionary runs. The left hand graph shows the maximum individual fitness across all 25 independent evolutionary runs, with a box plot every 5 generations to show the distribution. The right hand shows the distribution of fitnesses of the fittest individual, measured over 200 simulation and 20 real runs.

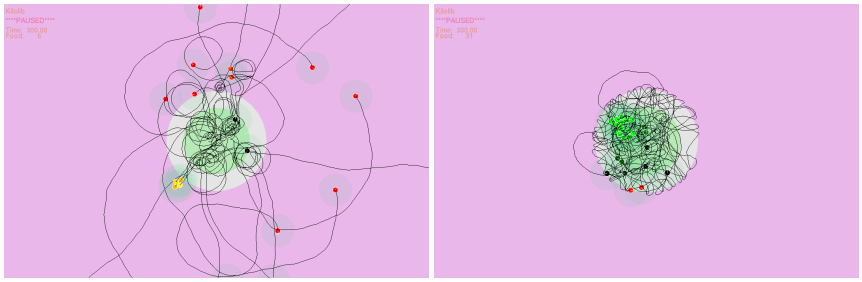

Fig. 4: Kilobot trails from simulation of the fittest controller in the first generation (left) and the 200th generation (right) of the fittest lineage.

We can examine the fittest BT, shown in Fig. 5, to gain insights into its workings. First of all, it is interesting to note that not all of the hardwired capabilities are used, only $detected\_food$, $\Delta dist_{food}$, and $\Delta dist_{nest}$. Both $scratchpad$ and $send\_signal$ are read but never written, so are equivalent to zero. This is not the case with all the evolved behaviour trees, see Table 4 for details of the blackboard usage of the top five fittest trees from different lineages. Between these individuals, every behaviour tree construct and blackboard entry is used. There is no obvious correlation between the features used and the fitness of the individual, perhaps indicating that there are multiple ways to solve this foraging problem.

		Blackboard entry									BT Nodes					
Rank	Fitness	1	2	3	4	5	6	7	8	9	SEQ	SEL	PROB	REPEAT	IF	SET
1	0.104				x				x	x	x	x		x	x	
2	0.0873		x	x	x						x	x	x		x	x
3	0.0853				x	x		x		x	x		x		x	x
4	0.0723	x	x		x	x	x	x	x	x	x	x	x	x	x	x
5	0.0710				x		x		x		x	x	x		x	

Table 4: Individuals from top five lineages and their usage of the blackboard and behaviour tree constructs. All individuals use at least the forward and one other of the motor action nodes. Usage is after redundant or unreachable nodes have been removed.

```
1  selm3(
2    seqm2(
3      ifgevar(send_signal, detected_food),
4      mf()),
5    seqm3(
6      seqm3(
7        ml(),
8        ifgevar(Δdist_food, scratchpad),
9        mf()),
10     ifgevar(Δdist_food, scratchpad),
11     seqm2(
12       seqm3(
13         seqm3(
14           ml(),
15           ifgevar(Δdist_nest, Δdist_nest),
16           mf(),
17         ifgevar(Δdist_food, send_signal),
18         mf()),
19       mf())),
20   seqm3(
21     ml(),
22     repeat(5,
23       ifltcon(Δdist_nest, −0.058530)),
24     ml()))
```

```
1  selm3(
2    seqm2(                    Move forward until in food
3      ifge(0, detected_food),
4      mf()),
5    seqm3(                    Turn and forward until out of food
6      ml(),
7      ifge(Δdist_food, 0),
8      mf(),
9      ml(),
10     mf(),
11     ifge(Δdist_food, 0),
12     mf(),
13     mf()),
14   seqm3(
15     ml(),
16     repeat(5,
17       iflt(Δdist_nest, −0.058530)),
18     ml()))
```

Fig. 5: Fittest behaviour tree. Left shows the code as evolved. Right shows the code with redundant lines removed by hand, the `seqm` nodes condensed, and conditionals simplified. Boxes highlight the three functional clauses.

The overall structure is a three-clause *selm*, the child trees will be *tick*ed in turn until one returns *success*. Consider a single kilobot, with no neighbours in communication with it. The first clause causes the kilobot to move forward as long as it is not in the food region. If it enters the food, the second clause comes into play, performing a series of left turns and forward movements until it moves out of the food region. Behaviour will then revert to the first clause and it will move forward again, likely hitting the nest region. We can see that this will produce reasonable individual foraging behaviour, and this pattern is visible in the right hand trail plot in Fig 4. The foraging behaviour will be enhanced in the presence of neighbours, since in this case the second clause will promote movement away from food generally, rather than just on the food region boundary. Finally, if the kilobot is executing

the second clause, manages to leave the food then re-enters it, or moves towards it in the presence of neighbours, the third clause is triggered, which produces some additional left turning. The *repeat* sub-clause will fail on the first iteration since it is not physically possible for the kilobot to move 59 mm in one update cycle of half a second.

This evolved behaviour tree is sufficiently small that it can be analysed by hand relatively easily. It may be that greater foraging performance could be obtained by removing the selective pressure to small trees, and a larger tree would be harder to analyse. But, in contrast to evolved neural networks, which are a black box for which there are no adequate tools to predict behaviour apart from direct testing [8], it is possible at least in principle to analyse any behaviour tree, in the same way it is possible to analyse any computer program. The behaviour of each sub-tree can be analysed in isolation, descending until the size of the sub-tree is tractable, and automatic tools can simplify and prune branches which will never be entered, or will always do nothing.

Understanding the behaviour of an evolved BT does not mean that it becomes possible to predict the emergent swarm behaviour that the interaction between the kilobots will produce. However, we believe that the more easily we can understand the controller, the more likely we are to gain insights into the problem of predicting these higher-level behaviours.

5 Conclusions and further work

Evolved controllers for swarm robotics are generally hard to understand. We have introduced the use of behaviour trees as an architecture for evolved swarm robot controllers that are more easily human readable. A simple foraging task was designed, a behaviour tree node set and blackboard interface specified, and a population of behaviour trees were evolved for a swarm of kilobot robots. The fittest individual was tested in real robots and showed good correspondence in performance to the individual in simulation. The individual was then analysed for insight into the discovered foraging algorithm.

There are many possible avenues for exploration in the application of genetic programming to behaviour trees since little work in this area exists. Choices of the evolutionary parameter values, and the filtering of environmental signals are somewhat arbitrary and will be explored further. The choice of blackboard and action nodes is another area for further investigation. We also want to develop automatic tools for simplifying the analysis of evolved trees.

We intend to apply the evolution of behaviour trees to other collective swarm robot tasks, using a more computationally capable platform that will not be so limited in possible tree size, and will also allow the on-board adaptive co-evolution of new BT controllers in response to changing environmental conditions. We are interested in the possibility of encapsulation of various swarm behaviours such as aggregation, flocking, and dispersion. In this, we are inspired by the argument of Francesca *et al.* [10] that restricting the representational power of the controller

allows the automatic discovery of solutions that are more resistant to reality gap effects, and feel the hierarchical structure of behaviour trees may lend themselves to tuning the bias-variance tradeoff.

Finally, we believe the increased human readability of evolved behaviour trees compared to other forms of evolved controller achieves progress towards more fully comprehending the emergence of collective behaviour from the interactions of individual agents.

References

1. Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, 7(1):1–41, 2013.
2. John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.
3. Erol Şahin. Swarm robotics: From sources of inspiration to domains of application. In *Swarm robotics*, pages 10–20. Springer, 2005.
4. Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. In *ACM SIGGRAPH Computer Graphics*, volume 21, pages 25–34. ACM, 1987.
5. Sabine Hauert, Jean-Christophe Zufferey, and Dario Floreano. Evolved swarming without positioning information: an application in aerial communication relay. *Autonomous Robots*, 26(1):21–32, 2009.
6. Sabine Hauert, J-C Zufferey, and Dario Floreano. Reverse-engineering of artificially evolved controllers for swarms of robots. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 55–61. IEEE, 2009.
7. Gianpiero Francesca and Mauro Birattari. Automatic design of robot swarms: achievements and challenges. *Frontiers in Robotics and AI*, 3:29, 2016.
8. Andrew L Nelson, Gregory J Barlow, and Lefteris Doitsidis. Fitness functions in evolutionary robotics: A survey and analysis. *Robotics and Autonomous Systems*, 57(4):345–370, 2009.
9. Stephane Doncieux, Nicolas Bredeche, Jean-Baptiste Mouret, and Agoston E Gusz Eiben. Evolutionary robotics: what, why, and where to. *Frontiers in Robotics and AI*, 2:4, 2015.
10. Gianpiero Francesca, Manuele Brambilla, Arne Brutschy, Vito Trianni, and Mauro Birattari. AutoMoDe: A novel approach to the automatic design of control software for robot swarms. *Swarm Intelligence*, 8(2):89–112, 2014.
11. Gianpiero Francesca, Manuele Brambilla, Arne Brutschy, Lorenzo Garattoni, Roman Miletitch, Gaëtan Podevijn, Andreagiovanni Reina, Touraj Soleymani, Mattia Salvaro, Carlo Pinciroli, et al. Automode-chocolate: automatic design of control software for robot swarms. *Swarm Intelligence*, 9(2-3):125–152, 2015.
12. Miguel Duarte, Sancho Moura Oliveira, and Anders Lyhne Christensen. Hybrid control for large swarms of aquatic drones. In *Proceedings of the 14th International Conference on the Synthesis & Simulation of Living Systems*, pages 785–792. Citeseer, 2014.
13. Miguel Duarte, Jorge Gomes, Vasco Costa, Sancho Moura Oliveira, and Anders Lyhne Christensen. *Hybrid Control for a Real Swarm Robotics System in an Intruder Detection Task*, pages 213–230. Springer International Publishing, Cham, 2016.
14. Petter Ogren. Increasing modularity of uav control systems using computer game behavior trees. In *AIAA Guidance, Navigation and Control Conference, Minneapolis, MN*, 2012.
15. Jeff Clune, Jean-Baptiste Mouret, and Hod Lipson. The evolutionary origins of modularity. *Proceedings of the Royal Society of London B: Biological Sciences*, 280(1755):20122863, 2013.
16. R Geoff Dromey. From requirements to design: Formalizing the key steps. In *Software Engineering and Formal Methods, 2003. Proceedings. First International Conference on*, pages 2–11. IEEE, 2003.

17. Damian Isla. Handling complexity in the halo 2 ai. In *Game Developers Conference*, volume 12, 2005.
18. Alex Champandard. Behavior trees for next-gen game ai. In *Game developers conference, audio lecture*, 2007.
19. Kevin Dill and L Martin. A game ai approach to autonomous control of virtual characters. In *Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC)*, 2011.
20. Alexander Shoulson, Francisco M Garcia, Matthew Jones, Robert Mead, and Norman I Badler. Parameterizing behavior trees. In *International Conference on Motion in Games*, pages 144–155. Springer, 2011.
21. Maria Cutumisu and Duane Szafron. An architecture for game behavior ai: Behavior multi-queues. In *AIIDE*, 2009.
22. J Andrew Bagnell, Felipe Cavalcanti, Lei Cui, Thomas Galluzzo, Martial Hebert, Moslem Kazemi, Matthew Klingensmith, Jacqueline Libby, Tian Yu Liu, Nancy Pollard, et al. An integrated system for autonomous robotics manipulation. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2955–2962. IEEE, 2012.
23. Rahib H Abiyev, ŞENOL Bektaş, Nurullah Akkaya, and Ersin Aytac. Behaviour trees based decision making for soccer robots. *Recent Advances in Mathematical Methods, Intelligent Systems and Materials*, 2013.
24. Andreas Klöckner. Interfacing behavior trees with the world using description logic. In *AIAA conference on Guidance, Navigation and Control, Boston*, 2013.
25. Michele Colledanchise and Petter Ogren. How behavior trees modularize robustness and safety in hybrid systems. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 1482–1488. IEEE, 2014.
26. Chong-U Lim, Robin Baumgarten, and Simon Colton. Evolving behaviour trees for the commercial game defcon. In *Applications of evolutionary computation*, pages 100–110. Springer, 2010.
27. Diego Perez, Miguel Nicolau, Michael O'Neill, and Anthony Brabazon. Evolving behaviour trees for the mario ai competition using grammatical evolution. In *Applications of evolutionary computation*, pages 123–132. Springer, 2011.
28. Alejandro Marzinotto, Michele Colledanchise, Colin Smith, and Petter Ogren. Towards a unified behavior trees framework for robot control. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 5420–5427. IEEE, 2014.
29. Renato de Pontes Pereira and Paulo Martins Engel. A framework for constrained and adaptive behavior-based agents. *arXiv preprint arXiv:1506.02312*, 2015.
30. Kirk YW Scheper, Sjoerd Tijmons, Cornelis C de Visser, and Guido CHE de Croon. Behavior trees for evolutionary robotics. *Artificial life*, 2015.
31. Michael Rubenstein, Christian Ahler, and Radhika Nagpal. Kilobot: A low cost scalable robot system for collective behaviors. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 3293–3298. IEEE, 2012.
32. Alan FT Winfield. Towards an engineering science of robot foraging. In *Distributed Autonomous Robotic Systems 8*, pages 185–192. Springer, 2009.
33. David C Hutchison. Introducing BrilliantColor™ Technology. *Texas Instruments white paper*, 2005.
34. Sabine Hauert, Laurent Winkler, Jean-Christophe Zufferey, and Dario Floreano. Ant-based swarming with positionless micro air vehicles for communication relay. *Swarm Intelligence*, 2(2-4):167–188, 2008.
35. Félix-Antoine Fortin, De Rainville, Marc-André Gardner Gardner, Marc Parizeau, Christian Gagné, et al. DEAP: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, 13(1):2171–2175, 2012.
36. Erin Catto. Box2D: A 2D Physics Engine for Games, 2008.
37. Nick Jakobi, Phil Husbands, and Inman Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. In *Advances in artificial life*, pages 704–720. Springer, 1995.
38. Simon Jones, Matthew Studley, and Alan Winfield. Mobile GPGPU Acceleration of Embodied Robot Simulation. In *Artificial Life and Intelligent Agents: First International Symposium, ALIA 2014, Bangor, UK, November 5-6, 2014. Revised Selected Papers*, Communications in Computer and Information Science. Springer International Publishing, 2015.